Let SQL Drive the XQuery Workhorse

(XQuery Join Graph Isolation)

Torsten Grust Manuel Mayr Jan Rittinger

Eberhard Karls Universität Tübingen
Tübingen, Germany

torsten.grust | manuel.mayr | jan.rittinger@uni-tuebingen.de

ABSTRACT

A purely relational account of the true XQuery semantics can turn any relational database system into an XQuery processor. Compiling nested expressions of the fully compositional XQuery language, however, yields odd algebraic plan shapes featuring scattered distributions of join operators that currently overwhelm commercial SQL query optimizers.

This work rewrites such plans before submission to the relational database back-end. Once cast into the shape of join graphs, we have found off-the-shelf relational query optimizers—the B-tree indexing subsystem and join tree planner, in particular—to cope and even be autonomously capable of "reinventing" advanced processing strategies that have originally been devised specifically for the XQuery domain, *e.g.*, XPath step reordering, axis reversal, and path stitching. Performance assessments provide evidence that relational query engines are among the most versatile and efficient XQuery processors readily available today.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing, Relational Databases*

1. INTRODUCTION

SQL query optimizers strive to produce query plans whose primary components are *join graphs*—bundles of relations interconnected by join predicates—while a secondary, peripheral *plan tail* performs further filtering, grouping, and sorting. Plans of this particular type are subject to effective optimization strategies that, taking into account the available indexes and applicable join methods, derive equivalent join trees, ideally with a left-deep profile to enable pipelining. For more than 30 years now, relational query processing infrastructure has been tuned to excel at the evaluation of plans of this shape.

SQL's rather rigid syntactical block structure facilitates its compilation into join graphs. The compilation of *truly compositional* expression-oriented languages like XQuery, however, may yield plans of unfamiliar shape [12]. The arbitrary nesting of `for` loops (iteration over ordered item sequences), in particular, leads to plans in which join and sort operators as well as duplicate elimination occur throughout. Such plans overwhelm current commercial SQL query optimizers: the numerous occurrences of sort operators block join operator movement, effectively separate the plan into fragments, and ultimately lead to unacceptable query performance.

Here, we propose a plan rewriting procedure that derives join graphs from plans generated by the XQuery compiler described in [12]. The XQuery order and duplicate semantics are preserved. The resulting plan may be equivalently expressed as a single `SELECT-DISTINCT-FROM-WHERE-ORDER BY` block to be submitted for execution by an off-the-shelf RDBMS. The database system then evaluates this query over a schema-oblivious tabular encoding of XML documents to compute the encoding of the resulting XML node sequence (which may then be serialized to yield the expected XML text).

In this work we restrict ourselves to the XQuery Core fragment, defined by the grammar in Fig. 1, that admits the orthogonal nesting of `for` loops over XML node sequences (of type `node()*`), supports the 12 axes of XQuery's *full axis* feature, arbitrary XPath name and kind tests, as well as general comparisons in conditional expressions whose `else` clause yields the empty sequence `()`. As such, the fragment is considerably more expressive than the widely considered

Expr	→	`for` $*VarName* `in` *Expr* `return` *Expr*	
	|	`let` $*VarName* `:=` *Expr* `return` *Expr*	
	|	$*VarName*	
	|	`if (`*BoolExpr*`) then` *Expr* `else ()`	
	|	`doc(`*StringLiteral*`)`	
	|	*Expr* `/` *ForwardAxis NodeTest*	
	|	*Expr* `/` *ReverseAxis NodeTest*	
BoolExpr	→	*Expr*	
	|	*Expr GeneralComp Expr*	
	|	*Expr GeneralComp Literal*	
GeneralComp	→	`=` | `!=` | `<` | `<=` | `>` | `>=`	[60]
ForwardAxis	→	`descendant::` | `following::` | ···	[73]
ReverseAxis	→	`parent::` | `ancestor::` | ···	[76]
NodeTest	→	*KindTest* | *NameTest*	[78]
Literal	→	*NumericLiteral* | *StringLiteral*	[85]
VarName	→	*QName*	[88]
StringLiteral	→	`"...."`	[144]

Figure 1: Relevant XQuery "workhorse" subset (source language). Annotations in [·] refer to the grammar rules in [4, Appendix A].

EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.

twig queries [6, 7] and can be characterized as XQuery's data-bound "workhorse": XQuery uses this fragment to collect, filter, and join nodes from participating XML documents before further processing steps (grouping, tree construction) are performed. Others have described the data needs of a query in terms of tree patterns [3], here we use a fragment of the XQuery language itself.

Isolating the join graph implied by the input XQuery expression lets the relational database query optimizer face a problem known inside out despite the source language not being SQL: in essence, the join graph isolation process emits a bundle of self-joins over the tabular XML document encoding connected by conjunctive equality and range predicates. Most interestingly, we have found relational query optimizers to be autonomously capable of translating these join graphs into join trees that, effectively, (1) perform cost-based shuffling of the evaluation order of XPath location steps and predicates, (2) exploit XPath axis reversal (*e.g.*, trade `ancestor` for `descendant`), and (3) break up and stitch complex path expressions. In recent years, all of these have been described as specific evaluation and optimization techniques in the XPath and XQuery domain [6, 16, 18]—here, instead, they are the *automatic result* of join tree planning solely based on the availability of vanilla B-tree indexes and associated statistics. The resulting plans fully exploit the relational database kernel infrastructure, effectively turning the RDBMS into an XQuery processor that can perfectly cope with large XML instances (of size 100 MB and beyond).

Join graph isolation is an essential part of *Pathfinder*[1]—a full-fledged compiler for the complete XQuery language specification that is targeting relational database back-ends. For all of these back-ends we observed significant query execution time improvements for popular XQuery benchmarks, *e.g.*, XMark or the query section of TPoX [17, 22]. One particular back-end is MonetDB. MonetDB/XQuery [5]—the combination of *Pathfinder* and MonetDB—uses the techniques described in this work to robustly detect value-based joins regardless of syntactic query variation. The role of join graph isolation becomes even more important in the upcoming version of MonetDB/XQuery: join graphs are the starting point for runtime query optimization [2].

We start to explore this form of XQuery join graph isolation in Section 2 where we review the compiler's algebraic target language, tabular XML document encodings, and join-based compilation rules for XPath location steps, nested `for` loops, and conditionals. The rewriting procedure of Section 3 then isolates the join graphs buried in the initial compiled plans. Cast in terms of an SQL query, IBM DB2 V9's relational query processor is able turn these graphs into join trees which, effectively, implement a series of otherwise XQuery- and XPath-specific optimizations. A further quantitative experimental assessment demonstrates that DB2 V9's built-in pureXML™ XQuery processor currently faces a serious challenger with its relational self if the latter is equipped with the join graph-isolating compiler (Section 4). Sections 5 and 6 conclude this paper with reviews of related efforts and work in flux.

2. JOIN-BASED XQUERY SEMANTICS

To prepare join graph isolation, the compiler translates the XQuery fragment of Fig. 1 into intermediate DAG-shaped plans over the logical table algebra of Table 1. This particularly simple algebra dialect has been designed to match the capabilities of SQL query engines: operators consume tables (not relations) and duplicate row elimination is explicit (in terms of δ). The row rank operator $\varrho_{a:\langle b_1,\ldots,b_n\rangle}$ exactly mimics SQL:1999's `RANK() OVER (ORDER BY` $b_1,\ldots,b_n$ `) AS` a and is primarily used to account for XQuery's pervasive sequence order notion. The attach operator $@_{a:c}(e)$ abbreviates $e \times \boxed{\begin{array}{c}a\\c\end{array}}$, where the right-hand side argument denotes a singleton literal table. Operator φ marks the serialization point at the root of the plan DAG, delivering those rows that encode the resulting XML node sequence. Below we will see how the join operator $\bowtie$ assumes a central role in the translation of XPath location steps, `for` loops, and conditional expressions.

[1] `http://www.pathfinder-xquery.org/`

Table 1: Table algebra dialect (compilation target language).

Operator	Semantics
φ_{b_1,b_2}	plan root (serialize column b_1 by order in b_2)
$\pi_{a_1:b_1,..,a_n:b_n}$	project onto columns b_i, rename b_i into a_i
σ_p	select rows that satisfy predicate p
$\_ \bowtie_p \_$	join with predicate p
$\_ \times \_$	Cartesian product
δ	eliminate duplicate rows
$@_{a:c}$	attach column a containing constant value c
$\#_a$	attach arbitrary unique row id in column a
$\varrho_{a:\langle b_1,..,b_n\rangle}$	attach row rank in a (in $b_1,\ldots,b_n$ order)
doc	XML document encoding table
$\boxed{\begin{array}{cc}a & b\\ c_1 & c_2\end{array}}$	singleton literal table (with columns a, b)

```
<open_auction id="1">
 <initial>
  15
 </initial>
 <bidder>
  <time>18:43</time>
  <increase>
   4.20
  </increase>
 </bidder>
</open_auction>
```

pre	size	level	kind	name	value	data
0	9	0	DOC	au···xml		
1	8	1	ELEM	open_···		
2	0	2	ATTR	id	1	1.0
3	1	2	ELEM	initial	15	15.0
4	0	3	TEXT		15	15.0
5	4	2	ELEM	bidder		
6	1	3	ELEM	time	18:43	
7	0	4	TEXT		18:43	
8	1	3	ELEM	incre···	4.20	4.2
9	0	4	TEXT		4.20	4.2

Figure 2: Encoding of the infoset of XML document `auction.xml`. Column `data` carries the nodes' typed decimal values.

2.1 XML Infoset Encoding

An encoding of persistent XML infosets is provided via the designated table `doc`. In principle, any schema-oblivious node-based encoding of XML nodes that admits the evaluation of XPath node tests and axis steps fits the bill (*e.g.*, *ORDPATH* [19]). The following uses one such row-based format in which, for each node v, key column `pre` holds v's unique document order rank to form—together with columns `size` (number of nodes in subtree below v) and `level` (length of path from v to its document root node)—an encoding of the XML tree structure (Fig. 2 and [13]). XPath kind and name tests access columns `kind` and `name`—multiple occurrences of value `DOC` in column `kind` indicate that table `doc` hosts several trees, distinguishable by their document URIs (in column `name`). For nodes with `size` $\leqslant 1$, table `doc` supports value-based node access in terms of two columns that carry

the node's untyped string value [9, § 3.5.2] and, if applicable, the result of a cast to type `xs:decimal`[2] (columns value and data, respectively). This tabular XML infoset representation may be efficiently populated (during a single parsing pass over the XML document text) and serialized again (via a table scan in pre order).

n	$kindt(n)$
`element(_,_)`	kind = ELEM
`attribute(_,_)`	kind = ATTR
`text()`	kind = TEXT
⋮	⋮

n	$namet(n)$
`element(`t`,_)`	name = t
`attribute(`t`,_)`	name = t
`text()`	*true*
⋮	⋮

α	$axis(\alpha)$
`child`	$\mathsf{pre}_\circ < \mathsf{pre} \leqslant \mathsf{pre}_\circ + \mathsf{size}_\circ \wedge \mathsf{level}_\circ + 1 = \mathsf{level}$
`descendant`	$\mathsf{pre}_\circ < \mathsf{pre} \leqslant \mathsf{pre}_\circ + \mathsf{size}_\circ$
`ancestor`	$\mathsf{pre} < \mathsf{pre}_\circ \leqslant \mathsf{pre} + \mathsf{size}$
`following`	$\mathsf{pre}_\circ + \mathsf{size}_\circ < \mathsf{pre}$
⋮	⋮

Figure 3: Predicates implementing the semantics of XPath kind and name tests—expressed in sequence type syntax [4, § 2.5.3]—and axes (excerpt). ∘ marks the properties of the context node(s).

2.2 XPath Location Steps

Further, this encoding has already been shown to admit the efficient join-based evaluation of location steps α`::`n along all 12 XPath axes α [13]. While the structural node relationship expressed by α maps into a conjunctive range join predicate $axis(\alpha)$ over columns pre, size, level, the step's kind and/or name test n yields equality predicates over kind and name (Fig. 3). Consider the three-step path Q_0 = `doc("auction.xml")/descendant::bidder/child::*/child::text()` over the document of Fig. 2. To perform the final `child::text()` step, which will have context elements `time` and `increase`, the database system evaluates a join between the document encoding and the step's context nodes (the query yields the pre ranks of the two resulting text nodes):

$$\pi_{\substack{\mathsf{item}:\\ \mathsf{pre}}}\left(\sigma_{\substack{kindt(\texttt{text()})\wedge\\ namet(\texttt{text()})}}(\mathsf{doc}) \underset{axis(\texttt{child})}{\bowtie} \begin{array}{|c|c|c|c|}\hline \mathsf{pre}_\circ & \mathsf{size}_\circ & \mathsf{level}_\circ & \cdots \\ \hline 6 & 1 & 3 & \vdots \\ 8 & 1 & 3 & \vdots \\ \hline \end{array}\right) = \begin{array}{|c|}\hline \mathsf{item} \\ \hline 7 \\ 9 \\ \hline \end{array}.$$

With their ability to perform range scans, regular B-tree indexes, built over table doc, perfectly support this style of location step evaluation [13].

2.3 A Loop-Lifting XQuery Compiler

From [12] we adopt a view of the dynamic XQuery semantics, *loop lifting*, that revolves around the `for` loop as the core language construct. *Any* subexpression e is considered to be iteratively evaluated inside its innermost enclosing `for` loop. For the XQuery fragment of Fig. 1, each iterated evaluation of e yields a (possibly empty) ordered sequence of nodes. To reflect this, we compile e into an algebraic plan that returns a ternary table with schema iter|pos|item: a row $[i, p, v]$ indicates that, in iteration i, the evaluation of e returned a sequence containing a node with pre rank v at sequence position p.

The inference rules Doc, Ddo, Step, If, ValComp, Comp, Let, For, and Var (taken from [12] and reproduced in Appendix A) form a *compositional* algebraic compilation scheme for the XQuery dialect of Fig. 1. The rule set expects to see the input query after XQuery Core normalization: the enforcement of duplicate node removal and document order after XPath location steps (via the application of `fs:distinct-doc-order(·)`, abbreviated to `fs:ddo(·)` in the following) and the computation of effective Boolean values in conditionals (via `fn: boolean(·)`) is explicit [9, § 4.2.1 and § 3.4.3].

Changing the XML infoset encoding in Section 2.1 or the XPath location step implementation in Section 2.2 requires a local modification only: the adjustment of the inference rules Doc and Step in Appendix A.[3]

2.4 The Compositionality Threat

To obtain an impression of typical plan features, we compile

```
doc("auction.xml")
  /descendant::open_auction[bidder] .        (Q1)
```

After XQuery Core normalization, this query reads

```
for $x in fs:ddo(doc("auction.xml")
                 /descendant::open_auction)
return if (fn:boolean(fs:ddo($x/child::bidder)))
       then $x else () .
```

Fig. 4 shows the initial plan for Q_1. Since the inference rules of Fig. 13 implement a *fully compositional* compilation scheme, we can readily identify how the subexpressions of Q_1 contribute to the overall plan (to this end, observe the gray plan sections all of which yield tables with columns iter|pos|item). XQuery is a functional expression-oriented language in which subexpressions are stacked upon each other to form complex queries. The tall plan profile with its stacked sections—reaching from a single instance of table doc (serving all node references) to the serialization point ⚲—directly reflects this orthogonal nesting of expressions.

Note, though, how this artifact of both, compositional language and compilation scheme, leads to plans whose shapes differ considerably from the ideal *join graph + plan tail* we have identified earlier. Instead, join operators occur in sections distributed all over the plan. A similar distribution can be observed for the blocking operators δ and ϱ (duplicate elimination and row ranking). This is quite unlike the algebraic plans produced by SQL `SELECT-FROM-WHERE` block compilation.

The omnipresence of blocking operators obstructs join operator movement and planning and leads industrial-strength optimizers, *e.g.*, IBM DB2 UDB V9, to execute the plan in stages that read and then again materialize temporary tables. In the following we will therefore follow a different route and instead *reshape* the plan into a join graph that becomes subject to efficient one-shot execution by the SQL database back-end. (Section 4 will show that join graph isolation for Q_1 improves the evaluation time by a factor of 5.)

[2] In the interest of space, we omit a discussion of the numerous further XML Schema built-in data types.

[3] MonetDB/XQuery, *e.g.*, uses a general structural join operator as a replacement for $\bowtie_{axis(\alpha)}$ in Rule Step. After join graph isolation, these structural join operators are transformed into physical loop-lifted staircase join primitives [5].

3. XQUERY JOIN GRAPH ISOLATION

In a nutshell, join graph isolation pursues a strategy that moves the blocking operators (ϱ and δ) into plan tail positions and, at the same time, pushes join operators down into the plan. This rewriting process will isolate a plan section, the *join graph*, that is populated with references to the infoset encoding table doc, joins, and further pipelineable operators, like projection, selection, and column attachment (π, σ, $@$).

The ultimate goal is to form a new DAG that may readily be translated into a *single* SELECT-DISTINCT-FROM-WHERE-ORDER BY block in which

(1) the FROM clause lists the required doc instances,
(2) the WHERE clause specifies a conjunctive self-join predicate over doc, reflecting the semantics of XPath location steps and predicates, and
(3) the SELECT-DISTINCT and ORDER BY clauses represent the plan tail.

3.1 Plan Property Inference

We account for the unusual tall shape and substantial size (of the order of 100 operators and beyond for typical benchmark-type queries) of the initial plan DAGs by a *peephole*-style rewriting process. For all operators $\circledast$, a property inference collects relevant information about the plan vicinity of $\circledast$. The applicability of a rewriting step may then decided by inspection of the properties of a single operator (and its closer neighborhood) at a time. Tables 2–5 define these properties and their inference in an operator-by-operator fashion. We rely on auxiliary function $cols(\cdot)$ that can determine the columns used in a predicate (*e.g.*, $cols(\mathsf{pre}_\circ + \mathsf{size}_\circ < \mathsf{pre}) = \{\mathsf{pre}_\circ, \mathsf{size}_\circ, \mathsf{pre}\}$) as well as the columns in the output table of a given plan fragment (*e.g.*, $cols\Big(@_{\mathsf{iter}:1}\big(\boxed{\mathsf{pos}\ \ 1}\big)\Big) = \{\mathsf{iter}, \mathsf{pos}\}$). Furthermore we use $\Rightarrow$ to denote the reachability relation of this DAG (*e.g.*, $\circledast \Rightarrow \varrho$ for any operator $\circledast$ in the plan).

icols This property records the set of input columns strictly required to evaluate $\circledast$ and its upstream plan. At the plan root ϱ, the property is initially seeded with the column set $\{\mathsf{pos}, \mathsf{item}\}$, the two columns required to represent and serialize the resulting XML node sequence. The *icols* columns are inferred during a top-down DAG walk. Among other uses, property *icols* facilitates *projection pushdown*—a standard rewriting technique in relational query processors [14].

const A set with elements of the form $a = c$, indicating that *all* rows in the table output by $\circledast$ hold value c in column a. Seeded at the plan leaves (*i.e.*, instances of doc or literal tables) and inferred bottom-up.

key The set of candidate keys generated by $\circledast$ (bottom-up). The more elaborate *key* inference rules for equi-joins ($\bowtie_{a=b}$) and rank operators (ϱ) follow from functional dependencies. [23, Section 5.2.1] gives a more detailed account of equi-join *key* inference.

set Boolean property *set* communicates whether the output rows of $\circledast$ will undergo duplicate elimination in the upstream plan. Inferred top-down (*set* is initialized to *true* for all operators but ϱ). Property *set* is a simpler and more modular representation of Rule 3 *Distinct Pushdown From/To* in [21].

3.2 Isolating Plan Tail and Join Graph

The isolation process is defined by three subgoals ⓔ, ⓓ, and ⊗ (described below), attained through a sequence of

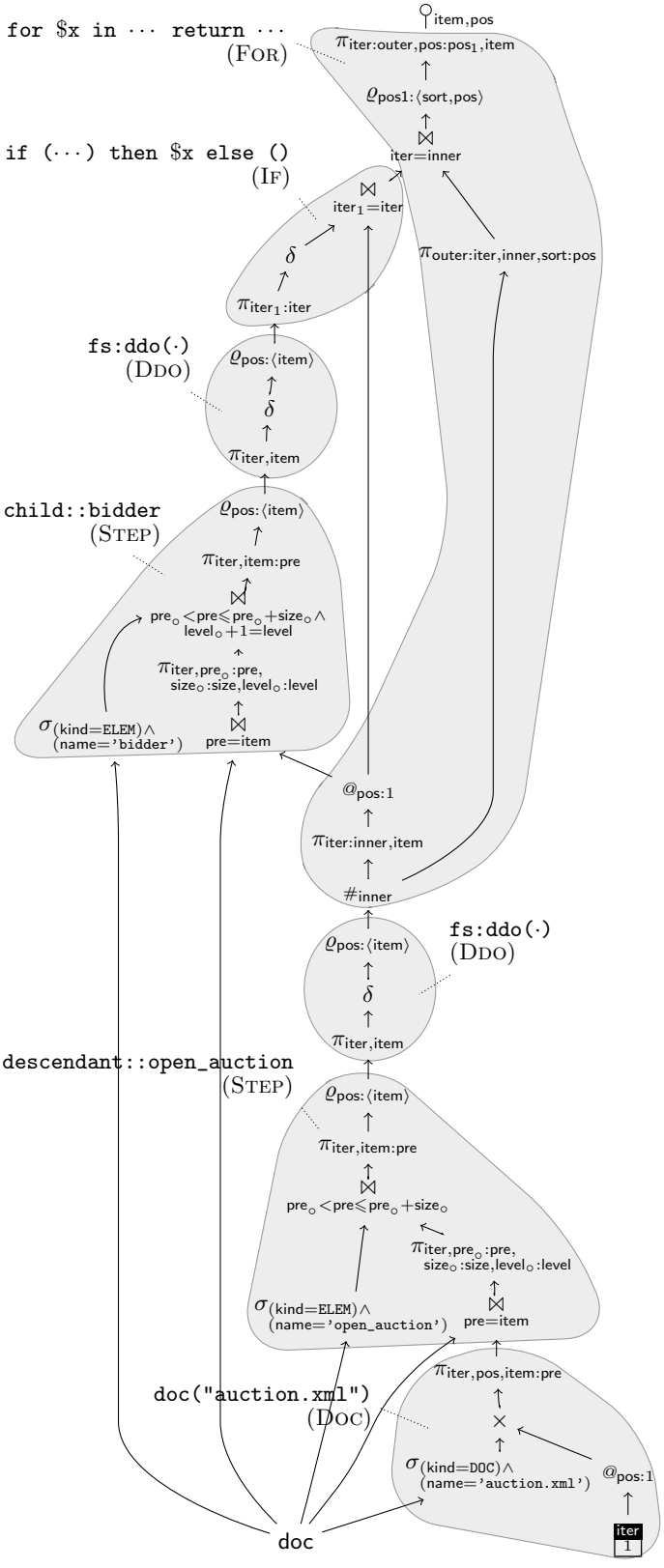

Figure 4: Initial stacked plan for Q_1. The gray plan sections indicate input XQuery subexpressions and applied compilation rules.

Table 2: Top-down inference of property *icols* for the input(s) of operator ⊛.

Operator ⊛	Inferred property *icols* of input(s) of ⊛
$\varrho_{b_1,b_2}(e)$	$e.icols \leftarrow \{b_1, b_2\}$
$\pi_{a_1:b_1,..,a_n:b_n}(e)$	$e.icols \leftarrow e.icols \cup \{b_i \mid a_i \in (icols \cap \{a_1, .., a_n\})\}$
$\sigma_p(e)$	$e.icols \leftarrow e.icols \cup icols \cup cols(p)$
$e_1 \bowtie_p e_2$	$e_{1,2}.icols \leftarrow e_{1,2}.icols \cup ((icols \cup cols(p)) \cap cols(e_{1,2}))$
$e_1 \times e_2$	$e_{1,2}.icols \leftarrow e_{1,2}.icols \cup (icols \cap cols(e_{1,2}))$
$\delta(e)$	$e.icols \leftarrow e.icols \cup icols$
$@_{a:c}(e)$	$e.icols \leftarrow e.icols \cup (icols \setminus \{a\})$
$\#_a(e)$	$e.icols \leftarrow e.icols \cup (icols \setminus \{a\})$
$\varrho_{a:\langle b_1,..,b_n\rangle}(e)$	$e.icols \leftarrow e.icols \cup (icols \setminus \{a\}) \cup \{b_1, .., b_n\}$
doc	–
a b / c_1 c_2	–

Table 3: Bottom-up inference of property *const* for operator ⊛.

Operator ⊛	Inferred property *const* of ⊛
$\varrho_{b_1,b_2}(e)$	$const \leftarrow e.const$
$\pi_{a_1:b_1,..,a_n:b_n}(e)$	$const \leftarrow \{a_i = c \mid (b_i = c) \in e.const\}$
$\sigma_p(e)$	$const \leftarrow e.const$
$e_1 \bowtie_p e_2$	$const \leftarrow e_1.const \cup e_2.const$
$e_1 \times e_2$	$const \leftarrow e_1.const \cup e_2.const$
$\delta(e)$	$const \leftarrow e.const$
$@_{a:c}(e)$	$const \leftarrow e.const \cup \{a = c\}$
$\#_a(e)$	$const \leftarrow e.const$
$\varrho_{a:\langle b_1,..,b_n\rangle}(e)$	$const \leftarrow e.const$
doc	$const \leftarrow \varnothing$
a b / c_1 c_2	$const \leftarrow \{a = c_1, b = c_2\}$

Table 4: Bottom-up inference of property *key* for operator ⊛.

Operator ⊛	Inferred property *key* of ⊛
$\varrho_{b_1,b_2}(e)$	$key \leftarrow e.key$
$\pi_{a_1:b_1,..,a_n:b_n}(e)$	$key \leftarrow \{\{a_i \mid b_i \in k\} \mid k \in e.key, k \subseteq \{b_1, .., b_n\}\}$
$\sigma_p(e)$	$key \leftarrow e.key$
$e_1 \bowtie_{a=b} e_2$	$key \leftarrow \{k_1 \mid \{b\} \in e_2.key, k_1 \in e_1.key\}$ $\cup \{k_2 \mid \{a\} \in e_1.key, k_2 \in e_2.key\}$ $\cup \{(k_1 \setminus \{a\}) \cup k_2 \mid \{b\} \in e_2.key, k_1 \in e_1.key, k_2 \in e_2.key\}$ $\cup \{k_1 \cup (k_2 \setminus \{b\}) \mid \{a\} \in e_1.key, k_1 \in e_1.key, k_2 \in e_2.key\}$
$e_1 \bowtie_p e_2$	$key \leftarrow \{k_1 \cup k_2 \mid k_1 \in e_1.key, k_2 \in e_2.key\}$
$e_1 \times e_2$	$key \leftarrow \{k_1 \cup k_2 \mid k_1 \in e_1.key, k_2 \in e_2.key\}$
$\delta(e)$	$key \leftarrow e.key \cup \{cols(e)\}$
$@_{a:c}(e)$	$key \leftarrow e.key$
$\#_a(e)$	$key \leftarrow e.key \cup \{\{a\}\}$
$\varrho_{a:\langle b_1,..,b_n\rangle}(e)$	$key \leftarrow e.key$ $\cup \{\{a\} \cup (k \setminus \{b_1, .., b_n\}) \mid k \in e.key, k \cap \{b_1, .., b_n\} \neq \varnothing\}$
doc	$key \leftarrow \{\{\text{pre}\}\}$
a b / c_1 c_2	$key \leftarrow \{\{a\}, \{b\}, \{a, b\}\}$

Table 5: Top-down inference of Boolean property *set* for the input(s) of operator ⊛.

Operator ⊛	Inferred property *set* of the input(s) of ⊛
$\varrho_{b_1,b_2}(e)$	$e.set \leftarrow false$
$\pi_{a_1:b_1,..,a_n:b_n}(e)$	$e.set \leftarrow e.set \wedge set$
$\sigma_p(e)$	$e.set \leftarrow e.set \wedge set$
$e_1 \bowtie_p e_2$	$e_{1,2}.set \leftarrow e_{1,2}.set \wedge set$
$e_1 \times e_2$	$e_{1,2}.set \leftarrow e_{1,2}.set \wedge set$
$\delta(e)$	$e.set \leftarrow e.set \wedge true$
$@_{a:c}(e)$	$e.set \leftarrow e.set \wedge set$
$\#_a(e)$	$e.set \leftarrow false$
$\varrho_{a:\langle b_1,..,b_n\rangle}(e)$	$e.set \leftarrow e.set \wedge set$
doc	–
a b / c_1 c_2	–

$$\frac{}{q \times \boxed{\begin{smallmatrix} a \\ c \end{smallmatrix}} \rightarrow @_{a:c}(q)}(1) \qquad \frac{}{\pi_{a_1,\ldots,a_n}\left(\pi_{b_1,\ldots,b_m}(q)\right) \rightarrow \pi_{a_1,\ldots,a_n}(q)}(2) \qquad \frac{\{a = c, b = c\} \subseteq const}{q_1 \bowtie_{a=b} q_2 \rightarrow q_1 \times q_2}(3)$$

$$\frac{a \notin icols}{@_{a:c}(q) \rightarrow q}(4) \qquad \frac{a \notin icols}{\varrho_{a:\langle b_1,\ldots,b_n\rangle}(q) \rightarrow q}(5) \qquad \frac{a \notin icols}{\#_a(q) \rightarrow q}(6) \qquad \frac{icols \neq \varnothing \qquad \{a_1, \ldots, a_n\} \setminus icols \neq \varnothing}{\pi_{a_1,\ldots,a_n}(q) \rightarrow \pi_{\{a_1,\ldots,a_n\} \cap icols}(q)}(7)$$

$$\frac{const \setminus \{b_1, \ldots, b_n\} \neq \varnothing}{\varrho_{a:\langle b_1,\ldots,b_n\rangle}(q) \rightarrow \varrho_{a:\langle b_1,\ldots,b_n\rangle \setminus const}(q)}(8) \qquad \frac{}{\varrho_{a:\langle b\rangle}(q) \rightarrow \pi_{a:b,cols(q)}(q)}(9) \qquad \frac{\circledast \in \{\sigma_p, \delta, @, \#\} \qquad a \notin cols(p)}{\circledast(\varrho_{a:\langle b_1,\ldots,b_n\rangle}(q)) \rightarrow \varrho_{a:\langle b_1,\ldots,b_n\rangle}(\circledast(q))}(10)$$

$$\frac{}{\pi_{a,c_1,\ldots,c_m}\left(\varrho_{a:\langle b_1,\ldots,b_n\rangle}(q)\right) \rightarrow \varrho_{a:\langle b_1,\ldots,b_n\rangle}\left(\pi_{b_1,\ldots,b_n,c_1,\ldots,c_m}(q)\right)}(11) \qquad \frac{\circledast \in \{\bowtie_p, \times\} \qquad a \notin cols(p)}{\varrho_{a:\langle b_1,\ldots,b_n\rangle}(q_1) \circledast q_2 \rightarrow \varrho_{a:\langle b_1,\ldots,b_n\rangle}(q_1 \circledast q_2)}(12)$$

$$\frac{}{\varrho_{a:\langle \ldots,b_i,\ldots\rangle}\left(\varrho_{b_i:\langle c_1,\ldots,c_m\rangle}(q)\right) \rightarrow \varrho_{a:\langle \ldots,b_{i-1},c_1,\ldots,c_m,b_{i+1},\ldots\rangle}\left(\varrho_{b_i:\langle c_1,\ldots,c_m\rangle}(q)\right)}(13)$$

$$\frac{set}{\delta(q) \rightarrow q}(14) \qquad \frac{const \setminus icols \neq \varnothing}{\delta(q) \rightarrow \delta\left(\pi_{cols(q) \setminus (const \setminus icols)}(q)\right)}(15) \qquad \frac{\neg set \qquad \circledast \notin \{\delta, \varrho\} \qquad k \in key \qquad k \subseteq icols}{\circledast(q) \rightarrow \delta\left(\pi_{icols}(\circledast(q))\right)}(16)$$

$$\frac{\circledast \in \{\pi, \sigma_p, @\} \qquad a \in cols(q_1)}{\circledast(q_1) \bowtie_{a=b} q_2 \rightarrow \circledast(q_1 \bowtie_{a=b} q_2)}(17) \qquad \frac{\circledast \in \{\bowtie_p, \times\} \qquad a \in cols(q_2)}{(q_1 \circledast q_2) \bowtie_{a=b} q_3 \rightarrow q_1 \circledast (q_2 \bowtie_{a=b} q_3)}(18) \qquad \frac{\{a\} \in key \qquad q_1 \Rrightarrow q_2 \qquad q_2 \Rrightarrow q_1}{q_1 \bowtie_{a=a} q_2 \rightarrow q_1}(19)$$

Figure 5: Join graph isolation transformation (for a rule *lhs* → *rhs*, the properties *icols*, *const*, *set*, and *key* denote the properties of *lhs*).

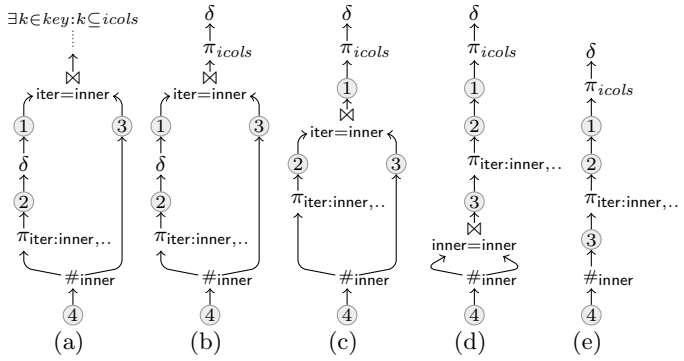

Figure 6: Moving duplicate elimination into the plan tail and join push-down (stages shown left to right).

goal-directed applications of the rewrite rules in Fig. 5.[4] Note how the rules' premises inspect the inferred plan properties, as described above. The subgoals either strictly move ϱ and δ towards the plan tail or push equi-joins ($\bowtie_{a=b}$) down into the plan. In addition to these three main goals, "house cleaning" is performed by rules defined to simplify or remove operators (Rules (1–8), (14), and (15)). House-cleaning in Fig. 4, *e.g.*, leads to the removal of the cross product and its rhs subtree (application of Rules (4)–(1)–(4)) as well as the removal of unreferenced rank operators (Rule (5)). While house-cleaning is performed whenever necessary, goal ⓔ is pursued before the two other subgoals.

ⓔ **Establish a single ϱ operator in the plan tail.** A *single* instance of the row ranking operator ϱ suffices to correctly implement the sequence and document order requirements of the overall plan. To this end, Rule (9) trades a ϱ for a projection if ϱ ranks over a single column. For the compilation Rules Ddo and Step, which introduce single-column row rankings ($\varrho_{\mathsf{pos}:\langle\mathsf{item}\rangle}$), this effectively means that document order determines sequence order—which is indeed the case for the result of XPath location steps and `fs:ddo(·)`. All other instances of ϱ ($\varrho_{\mathsf{pos}_1:\langle\mathsf{sort},\mathsf{pos}\rangle}$, introduced by For) are moved towards the plan tail via Rules (10–12). The premises of Rules (10) and (12) are no obstacle here: for the XQuery fragment of Fig. 1, the compiler does not emit predicates over sequence positions (column pos). Once arrived in the plan tail, Rule (13) splices the ranking criteria of adjacent ϱ operators. Rule (5) finally removes all but the topmost instance of ϱ.

ⓓ+ⓧ **Establish a single δ operator in the plan tail + join push-down and removal.** Duplicate elimination relocation and join push-down and removal are intertwined. Fig. 6—an abstract representation of Fig. 4—illustrates the stages of this process (the ◯ represent plan sections). These subgoals target and ultimately delete the equi-joins introduced by the compilation Rules Step, Comp, If, and For (the latter is in focus here).

A join of this type preserves the keys established by $\#_{\mathsf{inner}}$ and thus emits unique rows. The introduction of a new δ instance at the top of the plan fragment thus does not alter the plan semantics (Rule (16), see Figures 6(a) and (b)). This renders the original instance of δ obsolete as duplicate elimination now occurs upstream (Rule (14), Fig. 6(c)). The following stages push the join towards the plan base, leaving a trail of plan sections that formerly occurred in the join input branches (Rules (17–18), Figures 6(c) and (d)). Rule (19) detects that the join has degenerated into a key join over identical inputs and thus may be removed (Fig. 6(e)). Finally, this renders the remaining instance of $\#_{\mathsf{inner}}$ obsolete (column inner not referenced, Rule (6)).

Rewrite rules guarantee progress and termination. The house-cleaning rules (Rules (1–8), (14), and (15)) either remove an operator or restrict its arguments. The rules affecting rank operators (Rules (5) and (8–13)) pull-up rank operators through any other operator and result in at most a single rank operator at the plan root (see discussion of subgoal ⓔ). The compilation rules in Appendix A ensure that the position information forms a key at the plan root. Rule (16) thus ensures that a single distinct operator sits in the plan tail. With both distinct and rank operators out of the way, Rules (17–18) push down equi-joins until either an instance of # is reached or Rule (19) applies (see discussion on subgoal ⓧ). Once $\#_{\mathsf{inner}}$ becomes obsolete because column inner is no longer referenced by any equi-join, Rule (6) removes $\#_{\mathsf{inner}}$ and thus allows the remaining equi-joins to be pushed further done until no more rules are applicable.[5]

3.3 XQuery in the Guise of SQL SFW–Blocks

Fig. 7 depicts the isolation result for Query Q_1 (original plan shown in Fig. 4). The new plan features a bundle of operators in which—besides instances of π, σ—the only remaining joins originate from applications of compilation Rule Step, implementing the semantics of XPath location steps. The joins consume rows from the XML infoset encoding table doc which now is the only shared plan node in the DAG. As desired, we can also identify the plan tail (in the case of Q_1, no extra row ranking is required since the document order ranks of the elements resulting from the `descendant::open_auction` step—in column pre produced by the topmost π operator—already determine the overall order of the result.)

Quite unlike the initial plans emitted by the compositional compiler, XQuery join graph isolation derives logical query plans that are truly indistinguishable from the algebraic plans produced by a regular SQL translator. We thus let an off-the-shelf relational database back-end *autonomously* take over from here. It is now reasonable to expect the system to excel at the evaluation of the considered XQuery fragment as it will face a familiar workload. (This is exactly what we observe in Section 4.)

Most importantly, the join graphs provide a *complete* description of the input query's true XQuery semantics but *do not prescribe* a particular order of XPath location step or predicate evaluation. It is our intention to let the RDBMS decide on an evaluation strategy, based on its very own cost model, the availability of join algorithms, and supporting index structures. As a consequence, it suffices to communicate the join graph in form of a standard SQL `SELECT-DISTINCT-FROM-WHERE-ORDER BY`–block—*i.e.*, in a declarative fashion barring any XQuery-specific annotations or similar

[4] In the interest of succinctness, we omit the obvious variants of Rules (12), (17), and (18) in which the lhs and rhs arguments of the commutative operators $\bowtie_p$ and $\times$ are swapped, and ignore column renaming in Rules (2), (11), and (17).

[5] Note how adjacent equi-joins might lead to repeated applications of Rule (18). Our implementation avoids such repetition by taking operator argument plan sizes into account.

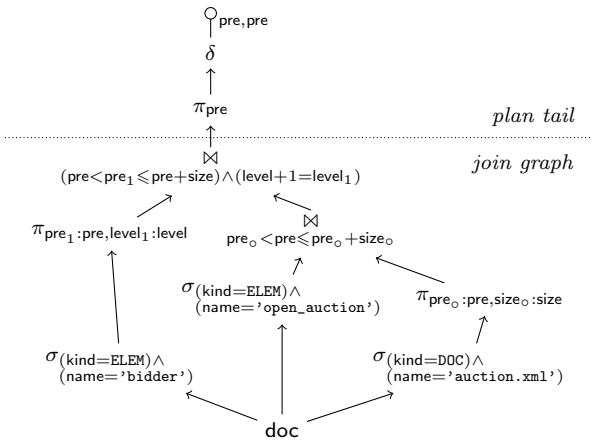

Figure 7: Final plan emitted for Q_1. The ····· separates the plan tail (above) from the isolated join bundle (three-fold self-join of table doc).

```
SELECT DISTINCT d2.pre
  FROM doc AS d1, doc AS d2, doc AS d3
 WHERE d1.kind = DOC
   AND d1.name = 'auction.xml'
   AND d2.kind = ELEM
   AND d2.name = 'open_auction'
   AND d2 BETWEEN d1.pre + 1 AND d1.pre + d1.size
   AND d3.kind = ELEM
   AND d3.name = 'bidder'
   AND d3 BETWEEN d2.pre + 1 AND d2.pre + d2.size
   AND d2.level + 1 = d3.level
 ORDER BY d2.pre
```

(Q_1^{SQL})

Figure 8: SQL encoding of Q_1's join graph.

clues. For Query Q_1, we thus ship the SQL query Q_1^{SQL} (Fig. 8) for execution by the database back-end.

Plan tail. The interaction of `for` loop iteration and sequence order of the final result becomes apparent in the plan tail of the following query (traversing XMark data [22] to return the names of those auction categories in which expensive items were sold at prices beyond $500):

```
let $a := doc("auction.xml")
for $ca in $a//closed_auction[price > 500],
    $i  in $a//item,
    $c  in $a//category
where $ca/itemref/@item = $i/@id
  and $i/incategory/@category = $c/@id
return $c/name
```

(Q_2)

XQuery Core normalization, compilation and subsequent isolation yields the SQL join graph query in Fig. 9 which describes a 12-fold self-join over table doc. Note how the `ORDER BY` and `DISTINCT` clauses—which represent the plan tail—reflect the XQuery sequence order and duplicate semantics:

Order The nesting of the three `for` loops in Q_2 principally determines the order of the resulting node sequence: in Q_2, row variables `d2`, `d4`, `d5` range over `closed_auction`, `item`, `category` element nodes, respectively. The document order of the `name` elements (bound to `d12`) is least relevant in this example and orders the nodes within each iteration of the innermost `for` loop.

Duplicates The XPath location step semantics requires duplicate node removal (row variable `d12` appears in the `DISTINCT` clause). Duplicates are retained, however, across `for` loop iterations (keys `d2.pre`, `d4.pre`, and `d5.pre` appear in the `DISTINCT` clause).

```
SELECT DISTINCT d12.pre,
       d2.pre AS item1, d4.pre AS item2,
       d5.pre AS item3
  FROM doc AS d1, ..., doc AS d12
 WHERE ...
 ORDER BY d2.pre, d4.pre, d5.pre, d12.pre
```

(Q_2^{SQL})

Figure 9: SQL encoding of Q_2 (focus on plan tail: order, duplicate removal).

Table 6: B-tree indexes proposed by `db2advis`.

Index key columns	Index deployment
nkspl nlkps nksp nlkp	XPath node test and axis step, access document node (`doc(·)`)
vnlkp nlkpv nkdlp	Atomization, value comparison with subsequent/preceding XPath step
p|nvkls	Serialization support (with columns nvkls in DB2's `INCLUDE(·)` clause [1])

p:pre, s:pre + size, l:level, k:kind, n:name, v:value, d:data

4. IN LABORATORY WITH IBM DB2 V9 (EXPERIMENTS)

The SQL language subset used to describe the XQuery join graphs—flat self-join chains, simple ordering criteria, and no grouping or aggregation—is sufficiently simple to let any SQL-capable RDBMS assume the role of a back-end for XQuery evaluation. We do *not* rely on SQL/XML functionality, in particular. In what follows, we will observe how IBM DB2 UDB V9 acts as a runtime for the join-graph-isolating compiler. In this context, DB2 V9 appears to be especially interesting, because the system

(1) has the ability to *autonomously* adapt the design of its physical layer, indexes in particular, in response to a given workload, and

(2) features the *built-in* XQuery processor pureXML™ which implements a "native" XML document storage and specific primitives for XPath evaluation, but nevertheless relies on the very same database kernel infrastructure. This will provide an insightful point of reference for the performance assessment of Section 4.2.

To provide the RDBMS with complete information about the expected incoming queries, we instructed the compiler to make the semantics of the serialization point ⚲ explicit—this adds one extra `descendant-or-self::node()` step to any Query Q, originating in its result node sequence:

`for $x in` Q `return $x/descendant-or-self::node()`.

This produces all XML nodes required to fully serialize the result (surfacing as the additional topmost self-join in the join plans of Figures 10 and 11).

Autonomous index design. For Q_2 as an representative of the prototypical expected query workload, the DB2 automatic design advisor, `db2advis` [1], suggests the B-tree index

Table 7: Relevant IBM DB2 plan operators.

Operator	Semantics	Operator	Semantics
RETURN	Result row delivery	SORT	Sort rows (+ dup. row elimination)
NLJOIN	Nested-loop join (left leg: outer)	HSJOIN	Hash join (left leg: probe)
IXSCAN	B-tree scan	TBSCAN	Temporary table scan
nlkps	Index access	doc	XML infoset table access

set of Table 6 (with a total size of 300 MB for a 110 MB instance of the XMark `auction.xml` document). Due to the regularity of the emitted SQL code, the utility of the proposed indexes will be high for any XQuery workload that exhibits a significant fraction of XQuery join graphs.

Partitioned B-tree index support for XQuery. The majority of the index keys proposed in Table 6 are prefixed with *low cardinality* column(s), *e.g.*, nk, or nlk. An XMark XML instance features 77 distinct element tag and attribute names, regardless of the document size. Similar observations apply to the XML node kinds and the typical XML document height. A B-tree that is primarily organized by such a low cardinality column will, in consequence, *partition* the XML infoset encoding into few disjoint node sets [11]. Note how, in a sense, a name-prefixed index key leads to a B-tree-based implementation of the *element tag streams*, the principal data access path used in the so-called *twig join* algorithms [6, 7].

The design advisor further suggests an index with key vnlkp whose value column prefix supports atomization and the general value comparisons between (attribute) nodes featured in Q_2. A B-tree of this type bears some close resemblance with the XPath-specific indexes (`CREATE INDEX ... GENERATE KEY USING XMLPATTERN ... AS SQL VARCHAR(`n`)`) employed by pureXML™ (Section 4.2).

4.1 XPath Continuations

How exactly does DB2 V9's query optimizer deploy the indexes proposed by its design advisor companion? An answer to this question can be found through an analysis of the plan trees generated by the optimizer. We have, in fact, observed a few not immediately obvious "tricks" that have found their way into the execution plans. Most of these observations are closely related to query evaluation techniques that have originally been described as XPath–specific [6, 16, 18], outside the relational domain. Since we have transferred all responsibility for the XQuery runtime aspects to the RDBMS, we think this is quite interesting.

The optimized DB2 execution plan found for Query Q_1 of Section 2.4 is shown in Fig. 10. We are reproducing these execution plans in a form closely resembling the output of DB2's visual explain facility. Nodes in these plans represent operators of DB2's variant of physical algebra—all operators relevant for the present discussion are introduced in Table 7.

Path stitching and branching. Consider the B-tree index with key nksp. Due to its nk prefix, this index primarily provides support for XPath name and kind tests. In the execution plan for Q_1, the index is used to access the requested document node (name = `'auction.xml'` ∧ kind = `DOC`). Additionally, however, the index delivers the infoset

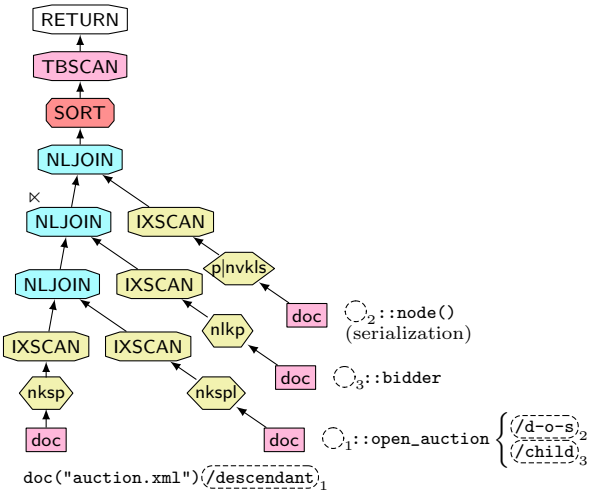

Figure 10: DB2 V9 execution plan for Q_1 with continuation annotations (`d-o-s` abbreviates `descendant-or-self`).

properties sp and thus provides all necessary information to step along the XPath `descendant` axis—namely the interval (pre, pre + size], see Fig. 3—from those nodes that have been found during index lookup. In the following, we will denote the result of such index lookups as

$$\texttt{doc("auction.xml")}\overset{\texttt{/descendant}}{\bigcirc}_1$$

or, generally $n\overset{/\alpha}{\bigcirc}_i$ (read: perform the specified node test n, then prepare a subsequent step along axis α). In the execution plan of Fig. 10, the bottom index nested-loop join continues this "half-cooked" step: a lookup in the index nkspl retrieves columns nkp to (1) perform the due name and kind test (name = `'open_auction'` ∧ kind = `ELEM`) and (2) complete the structural `descendant` axis traversal (check p for containment in the (pre, pre + size] interval obtained in the first half of the step). In the annotated plans, we write $\bigcirc_1$`::open_auction` (read: resume axis step and perform specified name and kind test). Stitched at the matching continuation points (here: those with subscript 1), we obtain the complete XPath location step again:

$$\texttt{doc("auction.xml")}\overset{\texttt{/descendant}}{\bigcirc}_1\texttt{::open\_auction}\ .$$

The lookup in index nkspl further provides the necessary infoset properties to prepare the now current continuations $\overset{\texttt{/descendant-or-self}}{\bigcirc}_2$ (columns sp) as well as $\overset{\texttt{/child}}{\bigcirc}_3$ (columns spl). Such continuations with multiple resumption points are the equivalent of the branching nodes discussed in the context of *holistic twig joins* [6].

Given the tailored B-tree index set in Table 6, the DB2 query optimizer consistently manages to select the index access path that provides just the required XML infoset properties. Resuming the `child` continuation at $\bigcirc_3$`::bidder` requires columns nk to perform the name and kind test plus columns pl to complete the evaluation of the range predicate that implements the `child` step. The semi-join evaluating this path step has its *early-out* flag set (see ⋉ in Fig. 10) and thus—similar to the original Query Q_1—applies only a predicate filter. Finally, as anticipated, the plan scans index p|nvkls to traverse all nodes in the subtrees below the query's XML result nodes (resuming from $\overset{\texttt{/descendant-or-self}}{\bigcirc}_2$).

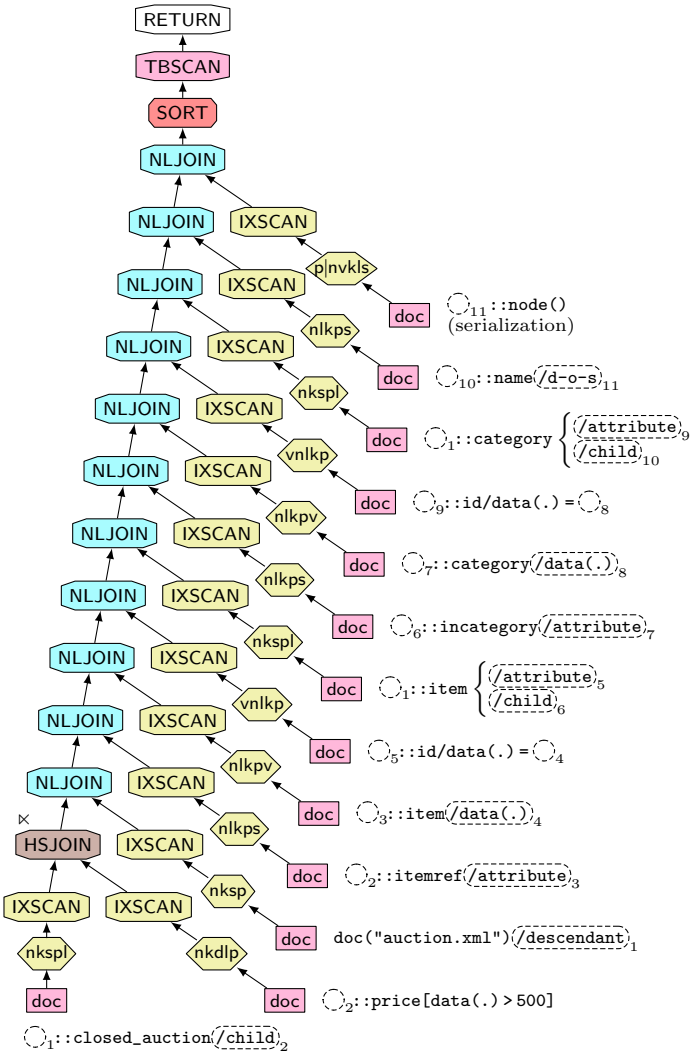

Figure 11: DB2 V9 execution plan for Q_2.

XPath step reordering and axis reversal. Two further phenomena in the execution plans can be explained in terms of the XPath continuation notion. For Q_1, the order of the XPath location steps specified in the input query did coincide with the join order in the execution plan (access document node of XML instance `auction.xml`, then perform a `descendant::open_auction` step, finally evaluate `child::bidder`). The B-tree index entries provide sufficient context information, however, to allow for arbitrary path processing orders—right-to-left strategies are conceivable as are strategies that start in the middle of a step sequence and then work their way towards the path's endpoints. The latter can be witnessed in the execution plan of Query Q_2 (Fig. 11).

The plan's very first index scan over nkspl evaluates the name and kind test for elements with tag `closed_auction` *before* the continuation for resumption point $\bigcirc_1$ has provided any context node information. The index key columns spl are used to prepare the continuation $\boxed{\texttt{/child}}_2$ which is then immediately resumed by the node test for element `price`. Index nkdlp is deployed to implement this node test as well as node atomization and subsequent value comparison (the index key contains column data of table doc). At this point, the plan has evaluated the path fragment

$$\bigcirc_1\texttt{::closed\_auction}\,\boxed{\texttt{/child}}_2\texttt{::price[data(.) > 500]}$$

which is still context-less. The due context is only provided by the subsequent NLJOIN–IXSCAN pair which verifies that the `closed_auction` elements found so far indeed are `descendant`s of `auction.xml`'s document node. Observe that in this specific evaluation order of the location steps, the `closed_auction` nodes now assume the context node role: the plan effectively determines the `closed_auction` elements that have the document node of `auction.xml` in their `ancestor` axis. The reversal of axes—in this case, trading `descendant` for `ancestor`—is based on the dualities pre $\leftrightarrow$ pre$_\circ$, size $\leftrightarrow$ size$_\circ$ in the predicates *axis*(`ancestor`) and *axis*(`descendant`), defined in Fig. 3. This observation applies to all other pairs of dual axes [18] (*e.g.*, `parent`/`child`, `following`/`preceding`) and, due to the attribute encoding used in table doc, also to the attribute/owner relationship between element nodes and the attributes they own.

The query optimizer decides on the reordering of paths and the associated reversal of XPath axes based on its "classical" selectivity notion and the availability of eligible access paths: for a B-tree with name-prefixed keys, the RDBMS's data distribution statistics capture tag name distribution while value-prefixed keys lead to statistics about the distribution of the (untyped) element and attribute values.

In the case of Q_2, this enabled the optimizer to decide that the access path nkdlp, directly leading to `price` nodes (key prefix nk) with a typed decimal value of greater than 500 (key column d), is highly selective (only 9,750 of the 4.7 million nodes in the 110 MB XMark XML instance are `price` elements and only a fraction of these has a typed value in the required range).

An analogous observation about the distribution of untyped string values in the value column—the key prefix in the vnlkp B-tree—has led the optimizer to evaluate the general attribute value comparison

$$\bigcirc_5\texttt{::id/data(.)} = \texttt{doc("auction.xml")/}\cdots\texttt{/attribute::item/data(.)}$$

before the "hole" $\bigcirc_5$ has been filled. The elements owning the `@id` attributes are resolved subsequently, effectively reversing the `attribute` axis. (This constellation repeats for the second attribute value comparison in Q_2, resumption point $\bigcirc_9$.)

4.2 Pure SQL vs. pureXML™

With DB2 Version 9, IBM released the built-in pureXML™ XQuery processor. This opens up a chance for a particularly insightful quantitative assessment of the potential of the purely relational approach to XQuery processing discussed here: not only can a comparison with pureXML™ be performed on the same machine, but even in the context of a single query processing infrastructure (implementation of query optimizer, plan operators, and B-tree indexes).

The tree traversal of XPath location steps in pureXML™ is implemented in terms of the new XSCAN low-level plan operator. The internals of XSCAN are based on the TurboXPath algorithm [15]. This is interesting in its own right: TurboXPath supports an XPath 2.0-style dialect quite similar to Fig. 1 and, in particular, admits nested `for` loops

Table 8: Sample query set taken from [15] (rightmost column shows the query identifier used in [15]). We replaced the non-standard `return-tuple` (Q_6) by an SQL/XML `XMLTABLE` construct.

	Query	Data	[15]
Q_3	`/site/people/person[@id = "person0"] /name/text()`	XMark	9a
Q_4	`//closed_auction/price/text()`	XMark	9c
Q_5	`/dblp/*[@key = "conf/vldb2001" and editor and title]/title`	DBLP	8c
Q_6	`for $thesis in /dblp/phdthesis [year < "1994" and author and title] return-tuple $thesis/title, $thesis/author, $thesis/year`	DBLP	8g

Table 9: Observed result sizes and wall clock execution times (average of 10 runs).

Query		DB2 + *Pathfinder*		DB2 pureXML™	
		stacked	**join graph**	whole	segmented
	# nodes	(sec)	(sec)	(sec)	(sec)
Q_1	1,625,157	63.011	**11.788**	10.073	9.661
Q_2	318	DNF	**0.544**	DNF	DNF
Q_3	1	60.582	**0.017**	0.891	0.001
Q_4	9,750	32.246	**0.309**	6.455	7.438
Q_5	1	442.745	**0.391**	48.066	0.001
Q_6	59	0.026	**0.004**	1.292	0.017

and XPath predicates, but does not implement the *full axis* feature (TurboXPath supports the vertical axes).

Aware of XSCAN's innards, to Q_1 and Q_2 we added sample queries that directly stem from [15]—these queries are displayed in Table 8. Each of these queries represents a much larger query set we investigated in the course of this work—together subsuming *e.g.*, all queries of the XMark and TPox benchmark sets [17, 22]. While Queries Q_1, Q_3, Q_4, and Q_5 are simple XPath queries, they differ in their runtime characteristics. Query Q_2 is a representative for queries with value-based joins (*e.g.*, XMark Queries 8–12). DB2's pureXML™ favors database designs that lead to comparably small XML document segments (of a few KB, say) per row. Since the tabular XML infoset encoding and XQuery processing strategy discussed here can perfectly cope with very large XML instances (beyond 100 MB), for the sake of comparison we let pureXML™ operate over both, segmented as well as monolithic XML documents.

The Queries Q_1–Q_4 ran against an XMark instance of 110 MB (4,690,648 nodes), Q_5, Q_6 queried an XML representation of Michael Ley's DBLP publication database (400 MB or 31,788,688 nodes). For pureXML's segmented document representation we cut the XMark instance into 23,000 segments of 1–6 KB and the DBLP instance into distinct publications, yielding about 1,000,000 segments of 30 nodes (ca. 800 bytes) per row. We further created an extensive `XMLPATTERN` index family such that at least one index was eligible to support the value references occurring in the query set (*e.g.*, for Q_3 we created an index on `/site/categories/category/@id`).

We then translated the query set with *Pathfinder*, an XQuery compiler that includes a faithful implementation of loop lifting and join graph isolation described in Sections 2.3 and 3. *Pathfinder* was configured to emit the SQL code derived from both, the original stacked plan and the isolated join graph. The resulting SQL queries ran against a database populated with tabular XML infoset encodings of the XMark and DBLP instances, using a B-tree index setup as described in Table 6. Both, pureXML™ and *Pathfinder* used the same DB2 UDB V9 instance hosted on a dual 3.2 GHz Intel Xeon™ computer with 8 GB of primary and SCSI-based secondary disk memory, running a Linux 2.6 kernel.

The impact of join graph isolation. Table 9 summarizes the average wall clock execution times we observed. Compositional compilation leads to tall stacked plans that exhibit a significant number of intermediate ϱ and δ operators. *Pathfinder* translates such plans into a SQL common table expression (`WITH...`) that features an equally large number of `DISTINCT` and `RANK() OVER (ORDER BY...)` clauses, leading to numerous SORT primitives in DB2 V9's execution plans. The application of join graph isolation to such plans can have drastic effects: for Query Q_1, for example, the join graph (Fig. 7) yields a five-fold reduction of execution time over the initial stacked plan (Fig. 4). Similarly, join graph isolation lets Q_2 execute in about 1/2 second (formerly the query did not complete within 20 hours).

Table 9 further assesses how DB2 pureXML™ fares against its *Pathfinder*-driven relational self. For Q_1, the universally high execution times of about 10 seconds largely reflect the substantial effort to support serialization: the resulting node sequence contains 3,249 `open_auction` elements, each being the root of a subtree containing 500 nodes on average. Query Q_4 primarily relies on raw path traversal performance as no value-based index can save the query engine from visiting a significant part of the XML instance. The more than 20-fold advantage of *Pathfinder* suggests that B-tree-supported location step evaluation will remain a true challenger for the XSCAN-based implementation inside pureXML™. Queries Q_3, Q_5 (and Q_6, to some extent) yield singleton (short) node sequences and constitute the best case for the segmented pureXML™ setup: here, `XMLPATTERN` index lookups return a single or few RID(s), directly leading the system to small XML segment(s)—the remaining traversal effort for XSCAN then is marginal. For the whole document setup, however, an index lookup could only point to the single monolithic XML instance: XSCAN thus does all the heavy work (the wildcard `*` in Q_5 forces the engine to scan the entire 400 MB DBLP instance). Despite the extensive index options available to support Q_2, pureXML™ is not able to finish evaluation within 20 hours: the system appears to miss the opportunity to perform value-based selections and joins early (recall the discussion of Fig. 11) and ultimately is overwhelmed by the Cartesian product of all `closed_auction`, `item`, and `category` elements. The indexes largely remain unused (the predicate `price[data(.) > 500]` is, in fact, evaluated second to last in the execution plan generated by pureXML™).

The sub-second execution times observed for *Pathfinder* indicate that the effort to compile into particularly simply-shaped self-join chains pays off. The DB2 V9 built-in monitor facility provides further evidence in this respect: the queries enjoy a buffer cache hit ratio of more than 90 % since merely table doc and indexes fight for page slots.

5. MORE RELATED WORK

One key ingredient in the join graph isolation process are the rewrites that move order maintenance and duplicate elimination into tail positions. Their importance is underlined by similar optimizations proposed by related efforts [10, 20, 24]: Fernández *et al.* remove order constraints and duplicates based on the XQuery Core representation [10]—an effect achieved by Rules (5) and (14) of Fig. 5. While their rewrites suffice to remove order and duplicate constraints in XPath queries like Queries Q_1, Q_3, Q_4, and Q_5, the principal data structure of XQuery Core—item sequences—prohibits merging of multiple orders as in Rule (13)—necessary to optimize, *e.g.*, Query Q_2. We might, however, still benefit from the optimization in [10], as it takes XPath step knowledge into account. We currently ignore this path step information and thus may end up ordering query results on more columns than strictly necessary.

In [24], an algebra over ordered tables is the subject of order optimization. An order context framework provides *minimal ordered semantics* by removing—much like Rule (5)—superfluous *Sortby* operators. In addition, order is merged in join operators and pushed through the plan in an *Orderby Pull up* much like in Section 3. In the presence of *order-destroying* operators such as δ, the technique of [24] however fails to propagate order information to the plan tail (cf. Rule (10)).

An extension of the tree algebra (TLC-C) in the research project Timber introduces order on a global level [20] and generates tree algebra plans. While Timber cannot handle item sequence order in general, the initial Timber-generated plans could be seen as an alternative representation of join graphs. Instead of turning the workload over to a relational database, however, Timber rewrites the plan to suit its own query engine which favors *early* duplicate elimination and sorting.

Most of the rewrites and property inference rules in the present work are an adaptation of techniques known already for more than a decade [14, 21, 23]. Representing order on a *logical* algebra level—in terms of the ranking operator ϱ—is a significant deviaton from previous efforts. We are not aware of further approaches that *encode order as data* and thus make order accessible to *logical* query optimization.

In the context of the MonetDB/XQuery open-source project[6], isolated join graphs form the starting point for the runtime optimization of XQuery expressions [2]: sampling and "zero-investment" algorithms detect data correlation and decide for the most efficient evaluation order at runtime. This approach further improves join tree planning and overcomes selectivity misestimation issues of classical optimizers.

In Section 4.1, we have seen how a selectivity-based reordering of XPath location steps can also lead to a reversal of axes. In effect, the optimizer mimics a family of rewrites that has been developed in [18]. These rewrites were originally designed to trade reverse XPath axes for their forward duals, which can significantly enlarge the class of expressions tractable by streaming XPath evaluators. Here, instead, we have found the optimizer to exploit the duality in both directions—in fact, a `descendant` axis step has been traded for an `ancestor` step in the execution plan for Q_2 (Fig. 11). The evaluation of rooted `/descendant::`n steps—pervasively introduced in [18] to establish a context node set of all elements with tag n in a document—is readily supported by the `n`-prefixed B-tree indexes. Since the XQuery compiler implements the *full axis* feature, it can actually realize a significant fraction of the rewrites in [18].

Although we exclusively rely on the vanilla B-tree indexes that are provided by any RDBMS kernel, cost-based join tree planning and join reordering leads to a remarkable plan versatility. In the terminology of [16], we have observed the optimizer to generate the whole variety of *Scan* (strict left-to-right location path evaluation), *Lindex* (right-to-left evaluation), and *Bindex* plans (hybrid evaluation, originating in a context node set established via tag name selection; cf. the initial `closed_auction` node test in Fig. 11).

The path branching and stitching capability (Section 4.1) makes the present XQuery compilation technique a distant relative of the larger family of *holistic twig join* algorithms [6, 7, 8]. We share the language dialect of Fig. 1—coined *generalized tree pattern* queries in [6, 8]—but add to this the *full axis* feature. Quite differently, though, we (1) let the RDBMS shoulder 100 % of the evaluation-time and parts of the compile-time effort invested by these algorithms (*e.g.*, the join tree planner implements the *findOrder*$(\cdot)$ procedure of [8] for free), and (2) use built-in B-tree indexes over table-shaped data where *TwigStack* [6] and *Twig*^{2}*Stack* [7] rely on special-purpose runtime data structures, *e.g.*, chains or hierarchies of linked stacks and modified B-trees, which call for significant invasive extensions to off-the-shelf database kernels.

6. WORK IN FLUX

This work rests on the maturity and versatility of database technology for strictly table-shaped data, resulting from 30+ years of experience. We (1) discussed relational encodings of the true XQuery semantics that are accessible for today's SQL query optimizers, but (2) also saw that some care is needed to unlock the potential of a set-oriented query processor.

The scope of the present work reaches beyond XQuery. Tall stacked plan shapes with scattered distributions of ϱ operators (Fig. 4) also are an artifact of the compilation of complex SQL/OLAP queries (in which functions of the `RANK()` family are pervasive). The observations of Section 4 suggest that the rewriting procedure of Fig. 5 can benefit commercial query optimizers also in this domain.

Acknowledgments. This research is supported by the German Research Council (DFG) under grant GR 2036/2-2.

7. REFERENCES

[1] DB2 9 for Linux, UNIX and Windows Manuals, 2007. `http://www.ibm.com/software/data/db2/udb/`.

[2] R. Abdel Kader, P. Boncz, S. Manegold, and M. van Keulen. ROX: run-time optimization of XQueries. In *Proc. SIGMOD*, 2009.

[3] A. Arion, V. Benzaken, I. Manolescu, and A. Pugliese. Structured Materialized Views for XML Queries. In *Proc. VLDB*, 2007.

[4] S. Boag, D. Chamberlin, and M. Fernández. XQuery 1.0: An XML Query Language. W3 Consortium, 2007. `http://www.w3.org/TR/xquery/`.

[5] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. SIGMOD*, 2006.

[6] `http://www.monetdb-xquery.org/`

$$\frac{}{\Gamma;\mathsf{loop}\vdash \texttt{doc}(uri)\Rrightarrow \pi_{\mathsf{iter,pos,item:pre}}\left(\sigma_{(\mathsf{kind}=\texttt{DOC})\wedge(\mathsf{name}=uri)}(\mathsf{doc})\times @_{\mathsf{pos}:1}(\mathsf{loop})\right)}(\textsc{Doc})\qquad \frac{\Gamma;\mathsf{loop}\vdash e\Rrightarrow q}{\Gamma;\mathsf{loop}\vdash \texttt{fs:ddo}(e)\Rrightarrow \varrho_{\mathsf{pos}:\langle\mathsf{item}\rangle}\left(\delta(\pi_{\mathsf{iter,item}}(q))\right)}(\textsc{Ddo})$$

$$\frac{\{\ldots,\$v_i\mapsto q_{v_i},\ldots\};\mathsf{loop}\vdash e_{if}\Rrightarrow q_{if}\qquad \mathsf{loop}_{if}\equiv\delta\left(\pi_{\mathsf{iter}_1:\mathsf{iter}}(q_{if})\right)\quad \left\{\ldots,\$v_i\mapsto \pi_{\mathsf{iter,pos,item}}(\mathsf{loop}_{if}\bowtie_{\mathsf{iter}_1=\mathsf{iter}} q_{v_i}),\ldots\right\};\mathsf{loop}_{if}\vdash e_{then}\Rrightarrow q}{\{\ldots,\$v_i\mapsto q_{v_i},\ldots\};\mathsf{loop}\vdash \texttt{if (fn:boolean}(e_{if})\texttt{) then}\, e_{then}\,\texttt{else ()}\Rrightarrow q}(\textsc{If})$$

$$\frac{\circledcirc\in\{\texttt{=},\texttt{!=},\texttt{<},\texttt{<=},\texttt{>},\texttt{>=}\}\qquad \Gamma;\mathsf{loop}\vdash e\Rrightarrow q}{\Gamma;\mathsf{loop}\vdash e\circledcirc val\Rrightarrow @_{\mathsf{item}:1}\left(@_{\mathsf{pos}:1}\left(\delta(\pi_{\mathsf{iter}}(\sigma_{\mathsf{value}\circledcirc val}(\mathsf{doc}\bowtie_{\mathsf{pre}=\mathsf{item}} q)))\right)\right)}(\textsc{ValComp})$$

$$\frac{\circledcirc\in\{\texttt{=},\texttt{!=},\texttt{<},\texttt{<=},\texttt{>},\texttt{>=}\}\qquad \Gamma;\mathsf{loop}\vdash e_1\Rrightarrow q_1\qquad \Gamma;\mathsf{loop}\vdash e_2\Rrightarrow q_2\quad q\equiv(\mathsf{doc}\bowtie_{\mathsf{pre}=\mathsf{item}} q_1)\bowtie_{\mathsf{iter}=\mathsf{iter}_1}\left(\pi_{\mathsf{iter}_1:\mathsf{iter},\mathsf{value}_1:\mathsf{value}}(\mathsf{doc}\bowtie_{\mathsf{pre}=\mathsf{item}} q_2)\right)}{\Gamma;\mathsf{loop}\vdash e_1\circledcirc e_2\Rrightarrow @_{\mathsf{item}:1}\left(@_{\mathsf{pos}:1}\left(\delta(\pi_{\mathsf{iter}}(\sigma_{\mathsf{value}\circledcirc\mathsf{value}_1}(q)))\right)\right)}(\textsc{Comp})$$

$$\frac{\Gamma;\mathsf{loop}\vdash e_{bind}\Rrightarrow q_{bind}\qquad \Gamma+\{\$x\mapsto q_{bind}\};\mathsf{loop}\vdash e_{ret}\Rrightarrow q}{\Gamma;\mathsf{loop}\vdash \texttt{let}\,\$x\,\texttt{:=}\,e_{bind}\,\texttt{return}\,e_{ret}\Rrightarrow q}(\textsc{Let})$$

$$\frac{\begin{array}{c}\{\ldots,\$v_i\mapsto q_{v_i},\ldots\};\mathsf{loop}\vdash e_{in}\Rrightarrow q_{in}\\ q_{\$x}\equiv \#_{\mathsf{inner}}(q_{in})\qquad \mathsf{map}\equiv\pi_{\mathsf{outer:iter,inner,sort:pos}}\ (q_{\$x})\\ \left\{\ldots,\$v_i\mapsto\pi_{\mathsf{iter:inner,pos,item}}\left(\mathsf{map}\bowtie_{\mathsf{outer}=\mathsf{iter}} q_{v_i}\right),\ldots\right\}+\\ \left\{\$x\mapsto @_{\mathsf{pos}:1}\left(\pi_{\mathsf{iter:inner,item}}(q_{\$x})\right)\right\};\pi_{\mathsf{iter:inner}}(\mathsf{map})\vdash e_{ret}\Rrightarrow q\end{array}}{\begin{array}{c}\{\ldots,\$v_i\mapsto q_{v_i},\ldots\};\mathsf{loop}\vdash \texttt{for}\,\$x\,\texttt{in}\,e_{in}\,\texttt{return}\,e_{ret}\Rrightarrow\\ \pi_{\mathsf{iter:outer,pos:pos}_1\mathsf{,item}}\left(\varrho_{\mathsf{pos}_1:\langle\mathsf{sort,pos}\rangle}\ (q\bowtie_{\mathsf{iter}=\mathsf{inner}}\mathsf{map})\right)\end{array}}(\textsc{For})\qquad \frac{}{\{\ldots,\$x\mapsto q,\ldots\};\mathsf{loop}\vdash\$x\Rrightarrow q}(\textsc{Var})$$

$$\frac{\Gamma;\mathsf{loop}\vdash e\Rrightarrow q}{\Gamma;\mathsf{loop}\vdash e/\alpha::n\Rrightarrow \varrho_{\mathsf{pos}:\langle\mathsf{item}\rangle}\left(\pi_{\mathsf{iter,item:pre}}\left(\sigma_{kindt(n)\wedge namet(n)}(\mathsf{doc})\bowtie_{axis(\alpha)}\left(\pi_{\mathsf{iter,pre}_\circ:\mathsf{pre,size}_\circ:\mathsf{size,level}_\circ:\mathsf{level}}\left(\mathsf{doc}\bowtie_{\mathsf{pre}=\mathsf{item}} q\right)\right)\right)\right)}(\textsc{Step})$$

Figure 13: Rules defining the compilation scheme $\Gamma;\mathsf{loop}\vdash e\Rrightarrow q$ from XQuery expression e to algebraic plan q.

[6] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. SIGMOD*, 2002.

[7] S. Chen, H. Li, J. Tatemura, W. Hsiung, D. Agrawal, and K. Selçuk Candan. Twig^{2}Stack: Bottom-Up Processing of Generalized-Tree-Pattern Queries over XML Documents. In *Proc. VLDB*, 2006.

[8] Z. Chen, H. Jagadish, L. Lakshmanan, and S. Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *Proc. VLDB*, 2003.

[9] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3 Consortium, 2007. `http://www.w3.org/TR/xquery-semantics/`.

[10] M. Fernández, J. Hidders, P. Michiels, J. Siméon, and R. Vercammen. Optimizing Sorting and Duplicate Elimination in XQuery Path Expressions. In *Proc. DEXA*, 2005.

[11] G. Graefe. Sorting and Indexing with Partitioned B-Trees. In *Proc. CIDR*, 2003.

[12] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. VLDB*, 2004.

[13] T. Grust, J. Teubner, and M. van Keulen. Accelerating XPath Evaluation in Any RDBMS. *ACM TODS*, 29(1), 2004.

[14] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 1984.

[15] V. Josifovski, M. Fontoura, and A. Barta. Querying XML Streams. *VLDB Journal*, 14(2), 2004.

[16] J. McHugh and J. Widom. Query Optimization for XML. *VLDB Journal*, 1999.

[17] M. Nicola, I. Kogan, and B. Schiefer. An XML Transaction Processing Benchmark. In *Proc. SIGMOD*, 2007.

[18] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Proc. XMLDM (EDBT Workshop)*, 2002.

[19] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. SIGMOD*, 2004.

[20] S. Paparizos and H. V. Jagadish. Pattern tree algebras: sets or sequences? In *Proc. VLDB*, 2005.

[21] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proc. SIGMOD*, 1992.

[22] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB*, 2002.

[23] D. E. Simmen, E. J. Shekita, and T. Malkemus. Fundamental Techniques for Order Optimization. In *Proc. SIGMOD*, 1996.

[24] S. Wang, E. A. Rundensteiner, and M. Mani. Optimization of Nested XQuery Expressions with Orderby Clauses. *DKE Journal*, 2006.

APPENDIX

A. INFERENCE RULES

The inference rule set of Fig. 13 (adopted from [12]) implements a loop-lifting XQuery compiler for the XQuery subset in Fig. 1 taking into account the XML encoding sketched in Section 2. The rule set defines a judgment

$$\Gamma;\mathsf{loop}\vdash e\Rrightarrow q\ ,$$

indicating that the XQuery expression e compiles into the algebraic plan q, given

(1) Γ, an environment that maps XQuery variables to their algebraic plan equivalent, and

(2) loop, a table with a single column iter that invariantly contains n arbitrary but distinct values if e is evaluated in n loop iterations.

An evaluation of the judgment $\varnothing;\boxed{\mathsf{iter}\\ 1}\vdash e_0\Rrightarrow q_0$ invokes the compiler for the top-level expression e_0 (the singleton loop relation represents the single iteration of a pseudo loop wrapped around e_0). The inference rules pass Γ and loop top-down and synthesize the plan q_0 in a bottom-up fashion. A serialization operator at the plan root completes the plan to read $\varrho_{\mathsf{item,pos}}(q_0)$.